Abstraction in Directed Model Checking

Stefan Edelkamp
Fachbereich Informatik
Universität Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Alberto Lluch-Lafuente
Dipartimento di Informatica
Università di Pisa
lafuente@di.unipi.it

Abstract

Abstraction is one of the most important issues to cope with large and infinite state spaces in model checking and to reduce the verification efforts. The abstract system is smaller than the original one and if the abstract system satisfies a correctness specification, so does the concrete one. However, abstractions may introduce a behavior violating the specification that is not present in the original system.

This paper bypasses this problem by proposing the combination of abstraction with heuristic search to improve error detection. The abstract system is explored in order to create a database that stores the exact distances from abstract states to the set of abstract error states. To check, whether or not the abstract behavior is present in the original system, efficient exploration algorithms exploit the database as a guidance.

Introduction

The ultimate goal of model checking (Clarke *et al.* 1999) is to prove the correctness of a system with respect to a given property. The correctness proof is often given through a complete exploration of the underlying state space. If the system does not satisfy the property, model checking algorithms return a witness of the error in form of a counterexample, which allows to locate and repair the faulty behavior. As a matter of fact, much of the success of model checking is due to its ability to find and report errors. The main drawback of model checking is the *state explosion problem*. State spaces of software designs are large enough in practice to make an exhaustive exploration infeasible. Moreover, many systems have an infinite state space. For instance, a simple program with only one integer variable may easily span the entire integer range. In this case, abstraction is one of the only chances to retain tractability.

It is not always easy to find abstractions that are correct with respect to the correctness specification and, at the same time, provide significant reductions. Exact approximations are abstractions that induce an abstract system, which is observationally equivalent to the original one; where the notion of observational equivalence depends, of course, on the logic used by specifications. They are difficult to be obtained and might result in weak reductions. Subsequently, research has mainly focused on over-approximations that induce *simulations* (Milner 1995), since they are found easier and provide more drastic reductions. Roughly speaking, system $\hat{M}$ simulates a system M if every behavior of M is also present in $\hat{M}$. If the abstraction applied induces a simulation, then the abstract system simulates the original one. As a consequence, if it satisfies the specification so does the original one. The opposite direction, however, is not true. A bad behavior in the abstract system might not be present in the original one. In this case we call the bad behavior a *spurious error*. Frequently, the approximation is refined for a new abstraction that is consistent with the counterexample established, and the verification process starts again. Our approach is different. If an error is found in the abstract system, we check for errors in the original system, using information collected in the exploration of the abstraction system as guidance.

Directed model checking incorporates heuristic search algorithms like A* to enhance the bug-finding capability of model checkers, by accelerating the search for errors and finding (near to) minimal counterexamples. In that manner we can mitigate the state explosion problem and the long counterexamples provided by some algorithms like DFS, which is often applied in explicit model checking.

One can distinguish about four main classes of evaluation functions based on the information they try to exploit. *Property specific* heuristics (Edelkamp *et al.* 2001) analyze the error description as the negation of the correctness specification. In some cases the underlying methods are only applicable to special kinds of errors. A heuristic that prioritizes transitions that block a higher number of processes focuses on deadlock detection. In other cases the approaches are applica-

ble to a wider range of errors. For instance, there are heuristics for invariant checking that extract information from the invariant specification and heuristics that base on already given errors states. The second class has been denoted as being *structural* (Groce and Visser 2004), in the sense that source code metrics govern the search. This class includes coverage metrics (such as *branch count*) as well as concurrency measures (such as *thread preference* and *thread interleaving*). Next there is the class of *user heuristics* that inherit guidance from the system designer in form of source annotations, yielding preference and pruning rules for the model checker. The last class are *planning heuristics* (Edelkamp 2003), where the problem of finding an error in the system is reduced to a planning task and solved using heuristic search action planners.

We introduce *abstraction databases* as a general paradigm to enhance the efficiency of directed model checkers. Our approach analyzes the abstract model in order to create a database which stores the shortest distance from a given abstract state to the set of abstract error states. This database is used as a heuristic during the exploration of the original system. As a result, proving that an abstract error is present in the original system can be done more efficiently. The resulting heuristics have a property, namely *monotonicity*, which guarantees optimal counterexamples if A* (Hart *et al.* 1968) is used as search algorithm. Another view of the paper's contribution is that while abstractions turned out to be a universal tool to verify a temporal property, they were only of limited help for falsifying it. With abstraction databases we utilize abstractions for improved falsification.

Our approach has a further advantage. It fits to both explicit and symbolic model checking approaches. Abstraction databases are constructed by exploring the abstract state space. As a consequence, this exploration can be done in a form of symbolic reachability analysis. The inferred symbolic representation of the set of reachable states together with the according distances to the goal, then serves as an estimate that can be referenced either in explicit and symbolic heuristic search.

The paper is structured as follows. First we introduce *pattern databases* as applied in AI search and give a theoretical foundation on *abstraction databases* based on Kripke structures. This will exploit similarities between the concepts developed in AI and model checking. Then we describe how abstraction databases can be applied in practice. We use the SPIN model checker and two of related tools, namely α-SPIN (for abstractions) and an HSF-SPIN (for heuristic search). We present a small set of experiments performed with such tools. Finally, we draw conclusions, discuss related work, and indicate future research avenues.

From Patterns to Abstractions

Pattern database search (Culberson and Schaeffer 1998) is an automatic technique for the improved design of *admissible* heuristics. Admissible heuristics are lower bounds on the distance to the set of goal states. This guarantees optimal goal paths and efficiency. A pattern database is a mere (hash) table, where indices are patterns and entries contain heuristic values. Patterns itself are simplified states according to a state relaxation function. The pattern database technique was first applied to define effective heuristics for sliding-tile puzzles. For this case, problem relaxation corresponds to removing a selected set of tiles from the board. The remaining set of tiles is referred to as the *pattern*. The *pattern database* stores all *pattern states* together with their shortest path distance on the simplified board to the pattern state for the goal. It is constructed in a BFS starting with the goal pattern and using inverse relaxed state transitions.

The idea of a pattern can be generalized as follows. If a state s is represented as a state vector $(s^1, \ldots, s^k)$ with variables s^i in some finite domains, then patterns are established by a projection, reducing the domains of some s^i, $i \in \{1, \ldots, k\}$. In a drastic, for example, the domain is reduced to the empty set, which entails ignoring the value of the variable.

In model checking this corresponds to a form of *data abstraction*, which exploits the fact that specifications for software models usually consider fairly simple relationships among the data values in the system. In such cases, one can map the domain of the actual data values into a smaller domain of abstract data values. Such mapping induces a mapping of the states of the system, which in turn induces an abstract system.

In many cases the abstract system simulates the original one (Clarke *et al.* 1999). Data abstraction is not the only approach. With *predicate abstraction* the concrete states of a system are mapped to abstract states according to their evaluation under a finite set of predicates (S. Graf and H. Saidi 1997). Automatic predicate abstraction approaches have been designed and implemented for finite and infinite state systems. The predicates $p_1, \ldots, p_n$ define an abstraction function $\phi : S \to \{0,1\}^n$ with $\phi(s) = \hat{s}$ if for all i we have $\hat{s}_i = p_i(s)$, that is abstract states refer to truth vectors of atomic propositions. Both data and predicate abstraction induce abstract systems that simulate the original one. We such an abstraction a *simulation abstraction*.

The common use of abstraction in practice consists

of a simple life cycle. First, the abstraction simulation is defined, according to the correctness property one wants to analyze. Then the model checker is used to verify the property on the abstract system. If the result is positive, then we know, that the original system satisfies the property. A negative answer, however, does not ensure that the original system violates the property too. There are various ways to proceed. Usually, a negative answer is accompanied with a counterexample, which is just a finite path that violates the property. An abstract path might not have a corresponding path in the concrete system. In this case, the path is called *spurious*. Hence, an abstract counterexample has to be analyzed in order to check whether it is spurious or not. One possible approach is to use the symbolic algorithm described in (Clarke *et al.* 2000), which basically consists of a forward search. Given an abstract counterexample path $\hat{s}_0, \ldots, \hat{s}_n$, the algorithm starts with the set of states given by $\phi^{-1}(\hat{s}_0)$. Then, it computes the set of successors of such states and performs the intersection with the set of states given by $\phi^{-1}(\hat{s}_1)$. The algorithm goes on until it establishes an empty set or $\phi^{-1}(\hat{s}_n)$.

The procedure is simpler if there is a clear correspondence between the abstract and the concrete system, where each code line in the abstract system corresponds to a line in the concrete system. As a consequence, each abstract transition has a corresponding concrete transition and one can just try to simulate the abstract counterexample in the original system (Pasareanu *et al.* 2001). If this is not possible, then the counterexample is spurious. Otherwise, the abstract counterexample is said to be feasible and one has to check whether it really violates the property.

Another approach is to analyze the *chose-free* state space of a system (Pasareanu *et al.* 2001). The idea is to bound non-deterministic branches of the state space, such that counterexamples are always *deterministic paths*. This approach uses a result, which states that every abstract non-deterministic path has a corresponding concrete path (Saidi 2000). If one of these methods certificates the existence of a spurious counterexample, the abstraction must be refined accordingly, and the previously described *abstract-check-refinement* cycle starts again.

In some cases, however, the strategy of validating abstract counterexamples is not suitable. We claim that an abstract counterexample might be the symptom of an error, even if it has no corresponding concrete counterexample. Consider the following trivial example for abstraction of a simple program: `x:=0; while x<n do x++; end do`. As an abstraction on variable x we take $\phi(0) =$ `ZERO`, $\phi(n) = N$, and $\phi(i) =$ `MIDDLE` for $i \in \{1, \ldots, n-1\}$, thus inducing the following abstract program:

```
x:=ZERO
while x&lt;N do
 if(x=0) then x=MIDDLE
 else x=nondeterministically MIDDLE or N
end do
```

In this trivial example, a state of the program is uniquely determined by the value of x. The original program has a finite path $(s_0, \ldots s_n)$, where s_i represents the state in which $x = i$. It is easy to see that ϕ is a simulation that induces an abstract system with infinitely many paths $(\phi(s_0), \phi(s_1)^j, \phi(s_n))$, where $j \geq 1$. Clearly, only the path given by $j = n-2$ has a corresponding concrete path. However, a model checker looking for an abstract state in which $x = n$, would find path $(\phi(s_0), \phi(s_1), \phi(s_n))$, which is a spurious counterexample. Classical approaches would try to refine the abstraction, while we propose to consider the path as a symptom for the existence of a similar path in the original system.

Abstraction Databases in Theory

For a formal treatment of abstraction databases we assume that the model is given in form of a Kripke structure. A *Kripke structure* $M = (S, L, AP, \rightarrow)$ consists of a set of states S, a set of atomic propositions AP which represent the observable properties of the system, a labelling function $L : S \rightarrow 2^{AP}$ that associate each state with the set of atomic propositions that hold in it, and a transition relation $\rightarrow \subseteq S \times S$. A set of initial states is often associated together with M. In the following we abbreviate $(s, t) \in \rightarrow$ with $s \rightarrow t$.

Given two Kripke structures $M = (S, L, AP, \rightarrow)$ and $\hat{M} = (\hat{S}, \hat{L}, \hat{AP}, \hat{\rightarrow})$ with $\hat{AP} \subseteq AP$, relation $\sim \subseteq S \times \hat{S}$ is a *simulation relation* between M and $\hat{M}$ whenever for all $s \sim \hat{s}$ we have $L(s) \cap \hat{AP} = \hat{L}(\hat{s})$ and for every state s_1 with $s \rightarrow s_1$ there is a state $\hat{s_1}$ and $\hat{s} \hat{\rightarrow} \hat{s_1}$ and $s_1 \sim \hat{s}_1$. We also say that $\hat{M}$ simulates M denoted by $M \preceq \hat{M}$.

It has been show that if $M \preceq \hat{M}$, then for every universal path quantified CTL* formula f with atomic propositions that are contained in $\hat{AP}$, $\hat{M} \models f$ implies $M \models f$ (Clarke *et al.* 1999). In addition, for every path $(s_0, s_1, \ldots)$ there is a corresponding path $(\hat{s}_0, \hat{s_1} \ldots)$ with $s_i \sim \hat{s_1}$, $i \geq 0$. So every behavior of M is also a behavior of $\hat{M}$.

An *abstraction* ϕ of a Kripke structure $M = (S, L, AP, \rightarrow)$ is a mapping from states to abstract states that induces an abstract Kripke structure $\hat{M} = (\hat{S}, \hat{L}, \hat{AP}, \hat{\rightarrow})$ as follows: $\hat{S} = \phi(S) = \{\phi(s) \mid s \in S\}$, and if $s \rightarrow s_1$ then $\phi(s) \hat{\rightarrow} \phi(s_1)$, i.e. $\hat{\rightarrow} =$

$\{(\phi(s), \phi(s_1)) \mid s \to s_1\}$. Generally speaking, we apply a surjection of the state space graph $(S, \to)$ into $(\hat{S}, \hat{\to})$. Abstraction ϕ is a homomorphism, since for all transitions $s \to s_1$ we have $\phi(s) \hat{\to} \phi(s_1)$. This implies that if state t is reachable from state s then $\phi(t)$ is reachable from $\phi(s)$.

We see that the definition of an abstraction is more restrictive than the definition of a simulation relation between M and $\hat{M}$ with $\hat{s} = \phi(s)$ and $\hat{s_1} = \phi(s_1)$. One difference is that in case of a simulation only the existence of a successor state $\hat{s_1}$ of s with $\hat{s}_1 \sim s_1$ is required, where here we fix $\hat{s_1}$ to be $\phi(s_1)$. We are not only interested in the existence of paths but also in shortest paths between states in the original and in abstract space.

Theorem 1 *Let M be a Kripke structure, ϕ be an abstraction, and $\hat{M}$ be the abstract structure under ϕ. For each two concrete states s and t we have that the shortest path from the abstract state $\phi(s)$ to $\phi(t)$ is not longer than the shortest path from s to t.*

Proof 1 *Let $p = (s = v_0, \dots v_n = t)$ be the shortest path form s to t in M. By homomorphism $\phi(p) = (\phi(v_0), \dots \phi(v_n))$ connects $\phi(s)$ to $\phi(t)$. The optimal path from $\phi(s)$ to $\phi(t)$ can only be shorter or equal.*

Recall that we focus on safety error detection, and that safety error detection can be reduced to invariant checking. Let f be the invariant. Error states are states in which f does not hold. Suppose that we have a set of error states T, then the abstract error is $\phi(T) = \{\phi(t) \mid t \in T\}$.

Theorem 1 suggests to use the shortest path in the abstract state space to the abstract error as an estimator in M. Hence, before checking for an error in the original system, we explore the abstract one to construct an abstraction database. An *abstraction database* according to an abstraction ϕ is a table with shortest path distances from each abstract state $\hat{s}$ to the abstract error set. While Theorem 1 proves the admissibility of the heuristic estimate we can also prove monotonicity.

Theorem 2 *Let M be a Kripke structure, ϕ be an abstraction, and $\hat{M}$ be the abstract structure under ϕ. The shortest path distance to the abstract error is monotone.*

Proof 2 *For each transition $s \to s_1$, homomorphism yields $\phi(s) \hat{\to} \phi(s_1)$. By the triangular inequality of shortest paths we have that 1 plus the shortest path from $\phi(s_1)$ to the abstract error is larger than or equal the shortest path from $\phi(s)$ to the abstract error.*

Theorem 2 is important, since monotonicity guarantees that A* variants can be implemented without any reopening strategy, which is more difficult to code and can lead to exponential running time with respect to the model size.

Until now, we have assumed that the state space graph for M is given explicitly. In practice, we have to explore it *on-the-fly* by successive enumeration. Therefore, defining $\hat{\to}$ by abstract state pairs is not sufficient, so that we also need to define abstract transitions to span abstract space. This brings us back to the simulation relation. If we define a mapping ϕ on the successor generation function, abbreviated as $\phi(\to)$, we require that each successor in abstract space has at least one successor in original space as a preimage. For Kripke structures $M = (S, L, AP, \to)$ this means that if $\phi(s)$ $\phi(\to)$ $\hat{s}_1$, then there exists s_1 with $s \to s_1$ and $\phi(s_1) = \hat{s}_1$. Now the definition matches the one given for a simulation relation. Therefore, the two aspects coincide. For abstractions that are simulations, Theorems 1 and 2 remain valid, so that the shortest path distance heuristic for abstract space is a monotone estimator function.

Consequently, we can use any abstraction that is a simulation relation to build an *abstraction database*. In database construction, the main limitation is the number of (abstract) states that can be hold in memory. Since databases are pre-computed, it is important to approximate an upper bound on the storage requirements each abstraction imposes. While there are approximations for the size of the abstract state spaces in the AI search domains, for general model checking we have not yet sufficient answers.

What is the running time for database construction? Backward breadth-first search starting from the set of abstract error states suffices to determine the state-to-goal distances in abstract space. This corresponds to linear time and space with respect to the size of the abstract model.

One of the main advantages of abstraction databases is that one can combine different abstractions to obtain a more informed heuristic. However, in order to get an admissible and monotone heuristic, each abtraction databases must be disjoint. Two abstraction databases according to abstractions ϕ and ψ are *disjoint*, if for all transitions t' in the abstract model for ϕ and transitions t'' in the abstract model for ψ we have $\phi^{-1}(t') \cap \psi^{-1}(t'') = \emptyset$. In other words, each operator introduces cost 1 in at most one abstraction, and cost 0 in all others. It $\phi^{-1}(t') \cap \psi^{-1}(t'') \neq \emptyset$ then the sum of the cost of t' and t'' in the respective abstract models is less than or equal to 1. Since in this case each operator is counted at most once, for each state we have that the sum of the shortest path distances from $\phi(s)$ and $\psi(s)$ to their respective abstract error state set is

still a lower bound to the shortest path from s to the error in the original space.

An example of the combination of abstraction databases is the FSM distance, which has been shown to be of practical use to shorten already established counterexamples (Edelkamp *et al.* 2001). Originally, this heuristic is defined as follows. Suppose the system consits on the asychronous composition of various processes given as communicating finite state machines. Suppose further that we have a state s' and we want to find the optimal path from an initial state to it. Since the composition is asynchronous, each system transition entails a transition in only one of the processes. Hence, in order for the system to reach state s' from state s each process must move from its local state in s to its local state in s'. The optimal local distance between process states can be efficiently computed before the verification starts. Then the distance between s and s' is estimated by the sum for all processes of the local optimal distances between the process states in s and in s'. Implictly, one is applying abstraction. The optimal distance between local states of a process i is a database corresponding to an abstraction of the system in which the value ofx every variable and every process except i is ignored. Implicitly, the FSM distance combines various disjoint abstraction databases, one for each process of the system.

The FSM distance heuristic is monotone and has been ported to program model checking to determine distances in Java byte- and C++ object-code (Leven *et al.* 2004).

Abstraction Databases in Practice

Nowadays, model checkers are frequently used for advanced AI planning, e.g. the Model-Based Planner (MBP) by Cimatti et al.[1] based on nuSMV has been applied for solving non-deterministic and conformant planning problems, including partial observable state variables and temporally extended goals. On the other hand, first model checking problems have been automatically converted to planning benchmarks[2]. Our empirical work takes the SPIN model checker (Holzmann 1997) as a practical case study model together with two tools: HSF-SPIN and α-SPIN.

SPIN takes a protocol specification written in the Promela language and produces source files that encode the state description and state transition function in native C code. These are linked together with the validator to allow exploration of the model. The user interface XSPIN allows to code the model, to run the validator with different parameters, to show the internal automata representation, and to simulate traces with message sequence charts.

α-SPIN (Merino *et al.* 2002) extends the SPIN validator by allowing the user to abstract the Promela model with suitable abstraction functions. The functions can be archived in a library for later reuse. The tool especially features *data abstraction* to the finite domain variables in the systems. The information flow is as follows. The model is scanned, processed with an XML parser, and simplified by the user selecting appropriate variables and abstraction functions, also coming in XML format. The result is another XML representation of the abstract model which, in turn, is translated back into Promela. Consequently, the approach produces no source conflict with SPIN.

The experimental HSF-SPIN model checker has been designed to allow different search algorithms to apply by providing a general state expanding subroutine, that generates a list of all successors of a system state. In its current implementation it provides *blind search algorithms* like depth-first search (DFS) and breadth-first search (BFS), *heuristic search algorithms* like best-first search, A* and IDA*, and *local search algorithms* like hill-climbing and genetic algorithms. The set of evaluation function ranges from *property specific* heuristics to *user* and *structural* ones. Moreover, exploration refinements like state compression, bit-state hashing, partial order and symmetry reduction have been successfully integrated to this algorithm portfolio. HSF-SPIN can handle a significant fraction of Promela and works with the same trace format as SPIN.

To use general abstraction functions we combined α-SPIN and HSF-SPIN as follows. First, HSF-SPIN has been extended to generate abstraction databases. With each state we attached a list of all predecessors on which a state is encountered. In fact this reflects the inverse of the underlying state space graph. In case of an error, the traversal is not terminated but the abstract error states are collected in a (priority) queue. Next, backward traversal is invoked on the implicitly represented inverse of the state space graph, starting with the queued set of abstract error states. Since we need estimates for each abstract state, we chose Dijkstra's single source all target shortest path algorithm. The established shortest path distances to the abstract error state are associated with each state in the hash table, yielding our *abstraction database*.

In the experiments we choose a problem that comes with the α-SPIN tool. By the novelty of the product the library of available abstractions and corresponding models is small. We took the model of an air condition system with n users in the room, everyone trying

[1] http://sra.itc.it/tools/mbp

[2] `http://ipc.icaps-conference.org`

heuristic	states	time	space
h_p	21,987	2.1s + 6.9s	8,449 KB
h_{p*}	381	13.0s + 0.01s	13 MB
$h \equiv 0$	26,087	6.6s	5,953 KB
h_m	25,195	6.6s	7,129 KB
h_f	25,936	7.1s	5.937 KB

Table 1: Using A* with abstraction database and other heuristics in the air condition model with $n = 2$. Abstraction databases time is split into file reading and pure search.

heuristic	states	time	space
h_p	149,818	22s + 1:07 min	35 MB
h_{p^*}	532	1:45 min + 2.2s	94 MB
$h \equiv 0$	183,475	1:01 min	24 MB
h_m	176,382	1:02 min	24 MB
h_f	182,961	1:07 min	24 MB

Table 2: Using A* with abstraction database and other heuristics in the air condition model with $n = 3$. Abstraction database time is split into file reading and pure search.

to change the temperature. We selected $n \in \{2, 3\}$. The model contained no bug, so that we had to seed one. We choose a simple assertion to be violated by the system in large depths. In the original system temperature ranges over integers. Heuristic (h_p) corresponds to an abstraction function that reduces the domain of the temperature. To illustrate the maximal reduction of our method and to indicate the range of different estimator functions that can be established, we also conducted experiments with an abstraction databases for the original model, which was possible, since the state spaces were small enough. Hence, heuristic $h*_p$ represents the actual optimal distance between system states. We compare the results of the pattern database heuristics with BFS ($h \equiv 0$) and with previously published heuristics like the formula-based heuristic (h_f) and the FSM distance (h_m) that takes a given error trail as an additional input. Tables 1 and 2 summarize the outcome of our small set of experiments. As expected, more space is needed to memorize the databases.

However, the abstraction database heuristics significantly reduced the number of states that are explored during search. Compared to the other heuristics the savings of h_p in the number of states are larger by a factor of about 5. For $n = 2$ an abstraction database was built, containing 15,858 states found while exploring the inverse graph structure. The initial state was re-encountered in depth 18 of 28 in total. For $n = 3$ the final databases had 28,782 entries, generated in 32 layers with the initial state re-established in depth 19. On our 248 MHz SUN Ultra the time to create the databases accumulated to about 10s for $n = 2$, and to about 6 min for $n = 3$. We observe a time slowdown in the exploration with respect to BFS and A* search and the other heuristic estimates. Time consumption for exploration increased from about 7s to about 9s ($n = 2$) and from about 1 min to about 1:30 min ($n = 3$). The profiler showed that almost all of the extra time is due to file scanning.

That we have not established a larger time gain during search is due to the additional time for the search in the (abstract) hash table. Twice as many calls to the hash table and state equality checks were needed. For $n = 2$ and BFS, we observed 124,463 table look-ups, while the abstraction database heuristic yields 222,925 calls. For $n = 3$ we found a discrepancy of about 1.2 to about 2.03 million calls.

For perfect databases (h_{p^*}) the exploration results were by far better. The exploration turned out to be optimal in the number of state expansions. Search was no longer harmed by additional hash table operations. With 1ms ($n = 2$) and 2.2s ($n = 3$) the pure search time was smaller than all other approaches by orders of magnitude. For $n = 3$ the construction of the perfect database with 805,760 entries took about 1.5 hours. For $n = 2$ the database with 113,072 entries was build in less than a minute.

Conclusions

This work combines abstraction reduction with heuristic search in order to improve the bug-finding capabilities of model checking. Automatically created heuristics in form of pre-computed heuristic databases are obtained by a backward traversal of the abstract state space. The entries are retrieved as estimate values error detection in the original system. The general scheme allows to refine heuristics for explicit and symbolic model checkers, where in this paper we concentrated on the explicit state model checking. Symbolic pattern databases for model checking are currently studied by (Nymeyer and Qian 2004). The abstraction process that comes along with every heuristic estimation can be automated and is suited to general model checking. The approach is not limited to data abstraction, any simulation with an abstract state space that fits into main memory can be used to enhance error detection. The new aspect is not only to consider the existence of behaviors in abstract space but also distance information.

Acknowledgements The first author is supported by DFG in the projects ED 74/2 and ED 74/3. The second author is supported by the European Research Training Network SEGRAVIS.

References

E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification (CAV)*, pages 154–169, 2000.

J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed model-checking in HSF-SPIN. In *Model Checking Software (SPIN)*, pages 57–79, 2001.

S. Edelkamp. Promela planning. In *Model Checking Software (SPIN)*, pages 197–212, 2003.

A. Groce and W. Visser. Heuristic model checking for Java programs. *Software Tools for Technology Transfer*, 2004.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for heuristic determination of minimum path cost. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Model Checking Software (SPIN)*, 2004.

P. Merino, M. del Mar Gallardo, J. Martinez, and E. Pimentel. aSpin: Extending spin with abstraction. In *Model Checking Software (SPIN)*, pages 254–258, 2002.

R. Milner. An algebraic definition of simulation between programs. In *Joint Conference of Artificial Intelligence (IJCAI)*, pages 481–489, 1995.

A. Nymeyer and K. Qian. Guided invariant model checking based on abastraction and symbolic pattern databases. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

C.S. Pasareanu, M. B.Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted Java programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 284–298, 2001.

S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, pages 72–83, 1997.

H. Saidi. Model checking guided abstraction and analysis. In *Static Analysis Symposium (SAS)*, pages 377–396, 2000.